Python comparison: Python uses `==` for value equality and doesn't do the same type of coercion.

Truthy and Falsy Values

```
// Falsy values
Boolean(0)         // false
Boolean('')        // false
Boolean(null)      // false
Boolean(undefined) // false
Boolean(NaN)       // false
Boolean(false)     // false

// Everything else is truthy, including:
Boolean([])        // true (empty array is truthy!)
Boolean({})        // true (empty object is truthy!)
Boolean('0')       // true (string with zero is truthy!)
```

Python comparison: In Python, empty containers like `[]` and `{}` are falsy.

Variable Hoisting and Scoping

Hoisting

```
// This works because of hoisting
x = 5;
console.log(x);  // 5
var x;

// The above is interpreted as:
var x;
x = 5;
console.log(x);  // 5

// But be careful with function-style declarations
console.log(y);  // undefined (not 5, only the declaration is hoisted)
var y = 5;

// let and const don't hoist the same way
console.log(z);  // ReferenceError
let z = 5;
```

Python comparison: Python doesn't hoist variables or functions.

Variable Scoping

```
// var is function-scoped
function varExample() {
    if (true) {
        var x = 10;
    }
    console.log(x);  // 10 (x is still accessible)
}

// let and const are block-scoped (more like Python)
function letExample() {
    if (true) {
        let y = 10;
    }
    console.log(y);  // ReferenceError: y is not defined
}
```

Python comparison: All variables in Python are block-scoped.

Functions and `this` Context

`this` Binding

```
const obj = {
    name: "Object",
    sayHello: function() {
        console.log(`Hello from ${this.name}`);
    }
};

obj.sayHello();  // "Hello from Object"

const func = obj.sayHello;
func();  // "Hello from undefined" (this is now the global context)

// Arrow functions don't have their own 'this'
const obj2 = {
    name: "Object2",
    sayHello: () => {
        console.log(`Hello from ${this.name}`);
    }
};

obj2.sayHello();  // "Hello from undefined" (this refers to parent scope)
```

Python comparison: Python uses explicit `self` parameter and doesn't have the concept of `this` binding changing based on how the function is called.

Default Parameters

```
// Default params
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}

// But watch out for falsy values
function badDefault(value = 10) {
    return value;
}

badDefault(0);  // Returns 0, not 10
badDefault('');  // Returns '', not 10
badDefault(null);  // Returns null, not 10
badDefault(undefined);  // Returns 10
```

Python comparison: Python's default parameters are similar but don't have issues with falsy values.

Function Arguments

```
function sum(a, b) {
    console.log(arguments);  // Arguments object contains all passed arguments
    return a + b;
}

sum(1, 2, 3, 4);  // Extra arguments are ignored but accessible via arguments

// Rest parameters (similar to Python's *args)
function betterSum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

betterSum(1, 2, 3, 4);  // 10
```

Python comparison: Python uses `*args` and `**kwargs` for flexible argument handling.

Arrays and Objects

Array Behavior

```
const arr = [1, 2, 3];

// Length can be modified directly
arr.length = 1;
console.log(arr);  // [1]

arr.length = 5;
console.log(arr);  // [1, empty × 4]

// Sparse arrays
const sparse = [];
sparse[0] = 1;
sparse[2] = 3;
console.log(sparse);  // [1, empty, 3]
console.log(sparse.length);  // 3

// forEach skips empty slots
sparse.forEach(x => console.log(x));  // Logs only 1 and 3
```

Python comparison: Python lists don't have sparse behavior and can't have their length directly modified.

Array Methods That Modify In-Place

```
const arr = [1, 2, 3];

// These modify the original array
arr.push(4);          // [1, 2, 3, 4]
arr.pop();            // [1, 2, 3], returns 4
arr.unshift(0);       // [0, 1, 2, 3]
arr.shift();          // [1, 2, 3], returns 0
arr.splice(1, 1, 5);  // [1, 5, 3], returns [2]
arr.sort();           // Sorts in-place
arr.reverse();        // Reverses in-place

// These create new arrays
const newArr = arr.map(x => x * 2);
const filtered = arr.filter(x => x > 1);
const sliced = arr.slice(1, 2);
```

Python comparison: Python differentiates between methods that modify in-place (like `list.sort()`) and functions that return new objects (like `sorted(list)`).

Object Quirks

```
// Object keys are always strings or Symbols
const obj = {
    1: 'one',
    true: 'yes'
};

console.log(obj[1]);      // 'one'
console.log(obj['1']);    // 'one' (same property)
console.log(obj[true]);   // 'yes'
console.log(obj['true']); // 'yes' (same property)

// Object property access
const key = 'name';
const person = {
    name: 'John'
};

console.log(person.name);    // 'John'
console.log(person['name']); // 'John'
console.log(person[key]);    // 'John'
```

Python comparison: Python dictionaries can have any hashable object as keys, not just strings.

Asynchronous JavaScript

Callbacks vs. Python's Sequential Execution

```
// Traditional callbacks
function fetchData(callback) {
    setTimeout(() => {
        callback('Data received');
    }, 1000);
}

fetchData(data => {
    console.log(data);  // Runs after 1 second
});
console.log('Continuing execution');  // Runs immediately
```

Python comparison: Python's synchronous code executes sequentially unless you use async/await or threading.

Promises

```
// Modern promise-based code
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Data received');
            // or reject(new Error('Failed to fetch data'))
        }, 1000);
    });
}

fetchData()
    .then(data => console.log(data))
    .catch(error => console.error(error));

console.log('Continuing execution');  // Runs before the promise resolves
```

Python comparison: Similar to Python's `asyncio` with `async`/`await`, but JavaScript has this pattern much more deeply embedded.

Async/Await

```
// Even more modern async/await
async function getData() {
    try {
        const data = await fetchData();  // fetchData returns a Promise
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

getData();
console.log('Continuing execution');  // Still runs before async function
completes
```

Python comparison: The syntax is very similar to Python's `async`/`await`.

Numeric Oddities

All Numbers are Floating Point

```
console.log(0.1 + 0.2);          // 0.30000000000000004
console.log(0.1 + 0.2 === 0.3);  // false

// Integer arithmetic can be imprecise for large numbers
console.log(9007199254740992 + 1); // 9007199254740992 (no change!)
```

Python comparison: Python handles this better with arbitrary precision integers and the Decimal module.

Special Number Values

```
console.log(1 / 0);              // Infinity
console.log(-1 / 0);             // -Infinity
console.log(0 / 0);              // NaN
console.log(NaN === NaN);        // false (NaN is never equal to anything)
console.log(Number.isNaN(NaN));  // true (proper way to check for NaN)
```

Python comparison: Python raises exceptions for divide by zero instead of returning Infinity.

Type Conversion Surprises

```
console.log(+'42');     // 42 (string to number)
console.log(+true);     // 1
console.log(+'abc');    // NaN

console.log(42 + '');   // '42' (number to string)
console.log({} + []);   // '[object Object]' (both convert to strings)

// The + operator
console.log(1 + 2);     // 3 (addition)
console.log('1' + '2'); // '12' (string concatenation)
console.log('1' + 2);   // '12' (number converts to string)
console.log(1 + '2');   // '12' (number converts to string)

// But other operators convert strings to numbers
console.log('5' - 2);   // 3
console.log('5' * 2);   // 10
console.log('5' / 2);   // 2.5
```

Python comparison: Python is much stricter and doesn't perform implicit type conversions.

Loop Peculiarities

`for...in` Loop Pitfalls

```
const arr = [1, 2, 3];
arr.customProp = 'surprise';

// for...in iterates over ALL enumerable properties, including inherited ones
for (const index in arr) {
    console.log(index, arr[index]);
}
// Outputs:
// '0' 1
// '1' 2
// '2' 3
// 'customProp' 'surprise'

// Use for...of for arrays (ES6+)
for (const value of arr) {
    console.log(value);  // Only outputs 1, 2, 3
}
```

Python comparison: Python's `for x in list` iterates over values, similar to JavaScript's `for...of`.

Object Iteration

```
const person = {
    name: 'John',
    age: 30
};

// ES6+ ways to iterate over objects
Object.keys(person).forEach(key => {
    console.log(key, person[key]);
});

Object.values(person).forEach(value => {
    console.log(value);
});

Object.entries(person).forEach(([key, value]) => {
    console.log(key, value);
});
```

Python comparison: In Python, `dict.items()`, `dict.keys()`, and `dict.values()` provide similar functionality.

Date and Time Weirdness

```
// Month is zero-indexed
const date = new Date(2023, 0, 15);  // January 15, 2023

// Date parsing is inconsistent across browsers
const parsed = new Date('2023-04-05');  // Different browsers might interpret
differently

// Date math
const today = new Date();
const tomorrow = new Date(today);
tomorrow.setDate(today.getDate() + 1);  // Add one day

// Date to timestamp
const timestamp = today.getTime();  // Milliseconds since Jan 1, 1970
```

Python comparison: Python's `datetime` module uses 1-indexed months and has more consistent parsing behavior.

The Module System

CommonJS (Node.js) vs. ES Modules

```
// CommonJS (Node.js)
const fs = require('fs');
module.exports = { myFunction };

// ES Modules (modern)
import fs from 'fs';
export const myFunction = () => {};
```

Python comparison: Python's import system is more straightforward and consistent.

Common Gotchas for Python Developers

Semicolons

```
// Semicolons are optional but recommended
const x = 5;  // With semicolon
const y = 10  // Without semicolon

// Automatic Semicolon Insertion can cause bugs
// This returns undefined because of ASI
function bad() {
    return
    {
        value: 42
    }
}

// Correct way
function good() {
    return {
        value: 42
    };
}
```

Python comparison: Python uses indentation for blocks and doesn't use semicolons.

Multiline Strings

```
// Template literals for multiline (ES6+)
const multiline = `This is a
multiline string
in JavaScript`;

// Old style concatenation
const oldMultiline = 'This is a \n' +
                     'multiline string\n' +
                     'in JavaScript';
```

Python comparison: Python supports triple-quoted strings for multiline text.

Global Scope Leakage

```
function leaky() {
    x = 5;  // Without var/let/const, becomes global variable!
}

leaky();
console.log(x);  // 5
```

Python comparison: Python doesn't create globals by default when variables are assigned in functions.

Prototype Inheritance vs. Class-based

```
// Old-style prototype inheritance
function Animal(name) {
    this.name = name;
}

Animal.prototype.speak = function() {
    console.log(`${this.name} makes a noise.`);
};

// Modern class syntax (syntactic sugar over prototypes)
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}
```

Python comparison: Python uses true class-based inheritance, not prototypes.

Event Loop and Callback Queue

```
console.log('First');

setTimeout(() => {
    console.log('Third');
}, 0);

console.log('Second');

// Output:
// First
// Second
// Third
```

Python comparison: Python's execution is typically synchronous unless explicitly using async features.

Debugging Tips for Python Developers

1. Use `console.log()` instead of `print()`.
2. Browser dev tools are your best friend (similar to Python's pdb).
3. Use `typeof` to check types: `typeof 42` → `'number'`.
4. Remember that objects and arrays are passed by reference.
5. Array indexes out of bounds return `undefined`, not errors.
6. Use `===` instead of `==` most of the time.
7. Watch out for scope issues with `var`.
8. Be careful with async code and callbacks.